Statistical learning (MT7038) - Project 1

Instructions: *This project consists of five tasks that should be solved individually. You are free to copy and modify code used in the project template file* ***MT7038_project1_VT19.Rmd*** *without further reference, but any code inspired by fellow students or other sources must be clearly acknowledged. Use of specific R-packages apart from those used in the template (the* ***tidyverse****-suite,* ***rpart*** *and* ***rpart.plot****) requires permission from course staff.*

The solution should be submitted at the course web page as a source markdown file together with a compiled pdf.

This project mainly aims at introducing a tidy (in the `tidyverse` sense) workflow in approaching statistical learning projects. There are R-packages designed to streamline such work, notably the `caret` package and tools under construction in `tidymodels`, but these tend to hide much detail under the hood. Instead, we will do much of the workflow-coding using data-manipulation tools from the `tidyverse`, in particular we will make excessive use of the `map*`-functions from `purrr` and list columns from `tibble`. If you are unfamiliar with the tools of the `tidyverse` suite of packages, you have most likely not participated in the prerequisite course Statistical data processing (MT5013) but argued that you have corresponding knowledge. Now is the time to show that you do. It is not required to use `tidyverse` tools in your solutions, all could be written in base R using loops or the `apply`-family of functions, but we will not offer a base R template.

```
library(tidyverse) # For data manipulation
library(rpart) # For trees in later part
library(rpart.plot) # For trees in later part
theme_set(theme_minimal()) # ggplot theme, set as you like
```

We will use the Prostate Cancer data described in section 3.2.1 of the textbook, where the aim is to predict level of prostate-specific antigen (`lpsa`) from a number of clinical measures. Data is available at the textbook webpage and downloaded by

```
# Import
url <- "https://web.stanford.edu/~hastie/ElemStatLearn/datasets/prostate.data"
prostate_raw <- read_tsv(url) %>% select(-X1)
glimpse(prostate_raw)
```

```
## Observations: 97
## Variables: 10
## $ lcavol  <dbl> -0.5798185, -0.9942523, -0.5108256, -1.2039728, 0.7514...
## $ lweight <dbl> 2.769459, 3.319626, 2.691243, 3.282789, 3.432373, 3.22...
## $ age     <dbl> 50, 58, 74, 58, 62, 50, 64, 58, 47, 63, 65, 63, 63, 67...
## $ lbph    <dbl> -1.3862944, -1.3862944, -1.3862944, -1.3862944, -1.386...
## $ svi     <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
## $ lcp     <dbl> -1.3862944, -1.3862944, -1.3862944, -1.3862944, -1.386...
## $ gleason <dbl> 6, 6, 7, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 6, 7, 6, ...
## $ pgg45   <dbl> 0, 0, 20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 30, 5, 5, 0, 30, ...
## $ lpsa    <dbl> -0.4307829, -0.1625189, -0.1625189, -0.1625189, 0.3715...
## $ train   <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, TRUE, FALSE...
```

As you can see, data has been split into a train and test set through the `train` column. We will keep this split and store the test set in a separate table for a final evaluation of model performance.

```
# Split into separate tables
prostate_raw_train <- prostate_raw %>%
    filter(train) %>%
    select(-train)
prostate_raw_test <- prostate_raw %>%
    filter(!train) %>%
    select(-train)
```

An important next step would be data exploration (of the training set) and preprocessing, we will skip most of this in the assignment and just

- standardise variables by removing mean and dividing by standard deviation (necessary for some methods, in particular for ridge regression that we will use).
- randomly reorder observations (always a good idea).

Always set a random seed to ensure your results can be reproduced.

```
set.seed(2019)

# Prepare by scaling and random reordering
col_mean <- map(prostate_raw_train, mean)
col_sd <- map(prostate_raw_train, sd)
prostate_prep_train <- prostate_raw_train %>%
    map2_df(col_mean, ~.x - .y) %>% # Remove mean
    map2_df(col_sd, ~.x / .y) %>% # Divide by sd
    slice(sample(1:n())) # Random reordering

prostate_prep_test <- prostate_raw_test %>%
    map2_df(col_mean, ~.x - .y) %>%
    map2_df(col_sd, ~.x / .y)
```

Task 1

In the above code, the mean and standard deviation of the training set was used to standardise the test set. Argue why this is a good idea.

Given a huge data set, we could split training data into a fixed training and validation set that can be used for model fitting and validation respectively. Ours is kind of small, and we will use cross validation to improve performance.

Cross-validation

The function `cv_fold` below splits data into `n_fold` folds and places the corresponding train/test-sets in a tidy data frame. This is our first use of list-columns, columns that are not atomic vectros. Each row will contain one fold and the `train` and `test` columns will contain the full train/test data for each fold (do not confuse this use of train/test with the separate tables constructed above). While this may not be very efficient in terms of memory storage, it is very convenient for further analysis.

```
cv_fold <- function(data, n_fold){
    # fold_id denotes in which fold the observation
    # belongs to the test set
    data <- mutate(data, fold_id = rep_len(1:n_fold, length.out = n()))
    # Two functions to split data into train and test sets
    cv_train <- function(fold, data){
        filter(data, fold_id != fold) %>%
            select(- fold_id)
    }
    cv_test <- function(fold, data){
        filter(data, fold_id == fold) %>%
            select(- fold_id)
    }
    # Folding
    tibble(fold = 1:n_fold) %>%
        mutate(train = map(fold, ~cv_train(.x, data)),
               test = map(fold, ~cv_test(.x, data)),
               fold = paste0("Fold", fold))
}
```

We now apply it to the training data using 10 folds, any further tuning should be based on the resulting `cv_prostate` data frame.

```
n_fold <- 10
cv_prostate <- cv_fold(prostate_prep_train, 10)
glimpse(cv_prostate)
```

```
## Observations: 10
## Variables: 3
## $ fold  <chr> "Fold1", "Fold2", "Fold3", "Fold4", "Fold5", "Fold6", "F...
## $ train <list> [<tbl_df[60 x 9]>, <tbl_df[60 x 9]>, <tbl_df[60 x 9]>, ...
## $ test  <list> [<tbl_df[7 x 9]>, <tbl_df[7 x 9]>, <tbl_df[7 x 9]>, <tb...
```

Make sure you understand the structure of `cv_prostate`!

Fitting a ridge regression model

We will now fit models to each of the rows of the `train` column and validate on the `test` column. In this example we will fit a ridge regression model using function `lm.ridge` below, where we have chosen to use R´s `formula` class to define response and covariates.

```
lm.ridge <- function(data, formula, lambda){
    # Given data, model formula and shrinkage parameter lambda,
    # this function returns a data.frame of estimated coefficients
    X <- model.matrix(formula, data) # Exctract design matrix X
    p <- ncol(X)
    y <- model.frame(formula, data) %>% # Extract vector of responses y
        model.extract("response")
    # Compute parameter estimates (Eq. (3.44) in textbook)
    R <- t(X) %*% X
    solve(R + lambda * diag(p)) %*% t(X) %*% as.matrix(y) %>%
        as.data.frame() %>%
```

```
        setNames("estimate") %>%
        rownames_to_column("variable")
}
```

We will also need a function that predicts the response given a new set of explanatory variables `newdata`, a model fit (parameter estimates) as returned by `lm.ridge` and the `formula` used for fitting the model.

```
predict.ridge <- function(newdata, fit, formula){
    model.matrix(formula, data = newdata) %*% as.matrix(fit$estimate) %>%
        as.numeric()
}
```

We will fit a model with all explanatory variables (denoted by `.`) and without an intercept (denoted by `-1`) since we have standardised the response variable `lpsa`

```
formula <- lpsa ~ -1 + .
```

and for a sequence of values for the hyperparameter λ

```
lambda_seq <- exp(seq(0, log(10), length.out = 10))
lambda_seq
```

```
##  [1]  1.000000  1.291550  1.668101  2.154435  2.782559  3.593814  4.641589
##  [8]  5.994843  7.742637 10.000000
```

The following few lines now fits the model to training data for each combination of *lambda* in `lambda_seq` and cross-validation fold (`model_fit` column), computes predictions for the corresponding test data (`predicted` column), extracts the observed response variables from the test sets (`actual` column) and finally computes the mean squared error `mse` as the mean squared difference between `actual` and `predicted`

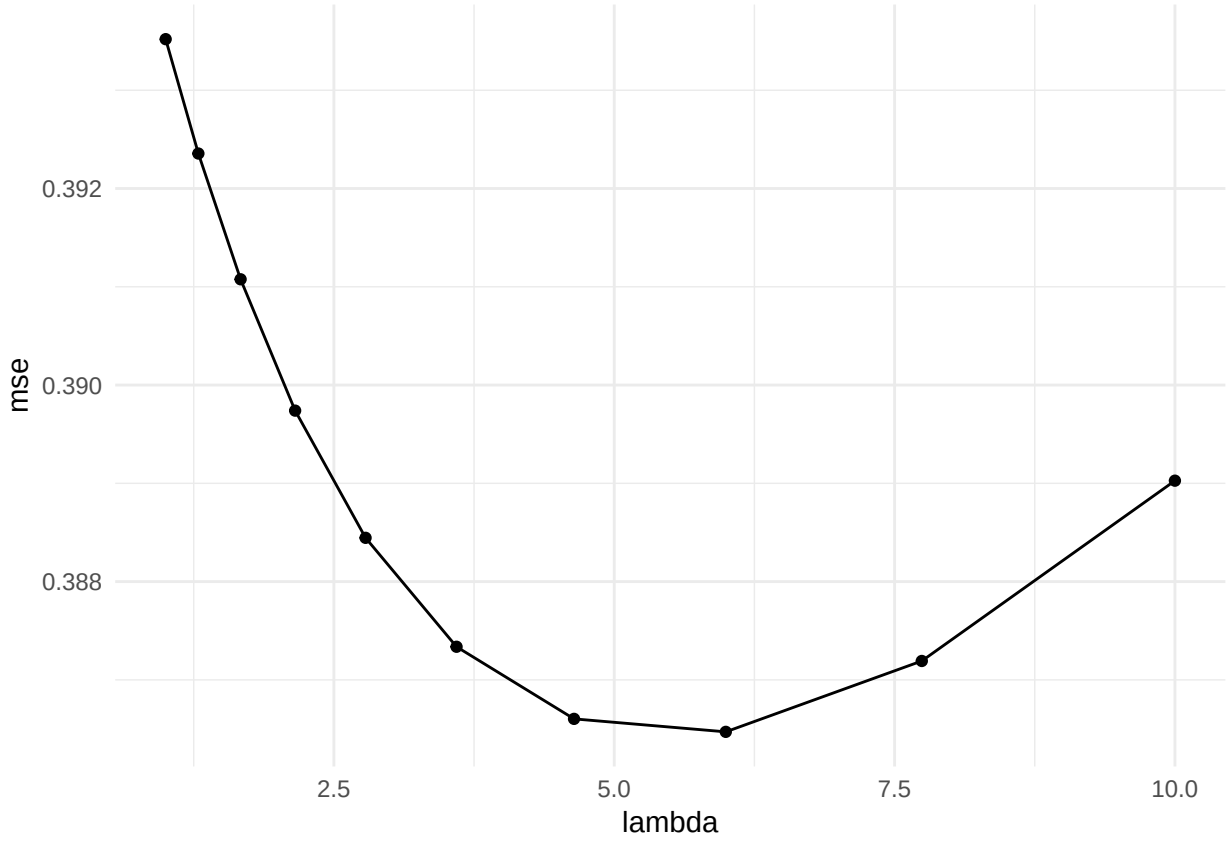
```
model_df <- cv_prostate %>%
    # One row for each combination of lambda and fold
    crossing(lambda = lambda_seq) %>%
    # Fit model to training data in each row
    mutate(model_fit = map2(train, lambda, ~lm.ridge(.x, formula, .y)),
           # Compute predicted valuea on test data
           predicted = map2(test, model_fit, ~predict.ridge(.x, .y, formula)),
           # Extract actual values from test data
           actual = map(test, ~(model.frame(formula, .x) %>%
                                    model.extract("response"))),
           # Compute mse
           mse = map2_dbl(predicted, actual, ~mean((.x - .y)^2)))
glimpse(model_df)
```

```
## Observations: 100
## Variables: 8
## $ fold      <chr> "Fold1", "Fold1", "Fold1", "Fold1", "Fold1", "Fold1"...
## $ train     <list> [<tbl_df[60 x 9]>, <tbl_df[60 x 9]>, <tbl_df[60 x 9...
## $ test      <list> [<tbl_df[7 x 9]>, <tbl_df[7 x 9]>, <tbl_df[7 x 9]>,...
## $ lambda    <dbl> 1.000000, 1.291550, 1.668101, 2.154435, 2.782559, 3....
## $ model_fit <list> [<data.frame[8 x 2]>, <data.frame[8 x 2]>, <data.fr...
```

```
## $ predicted <list> [<1.0004896, 0.6341363, 0.9496016, -0.4036718, -0.3...
## $ actual    <list> [<0.8140785, 0.4225387, 1.6002505, -0.3677154, -0.6...
## $ mse       <dbl> 0.3074743, 0.3096488, 0.3123780, 0.3157832, 0.320005...
```

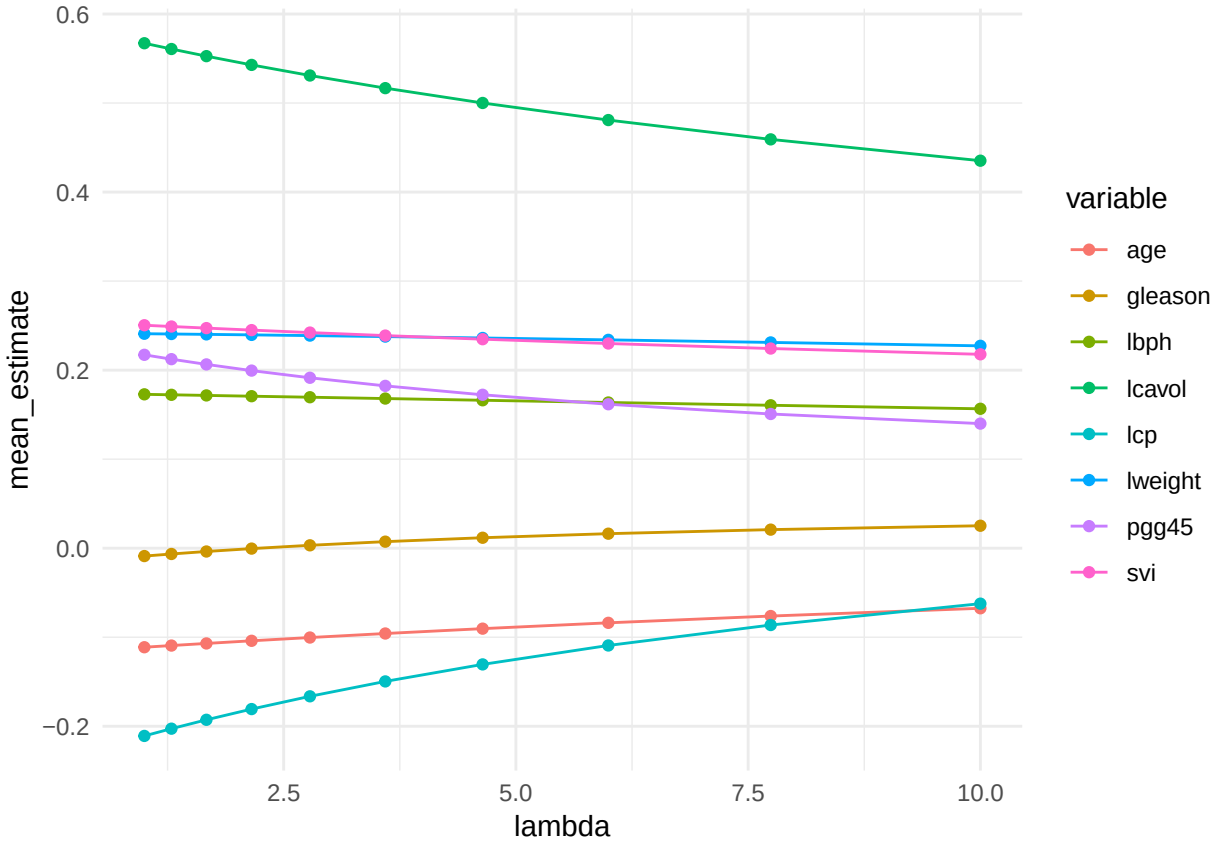
We are now ready to illustrate e.g. mean squared error as a function of hyperparameter by averaging over the folds for exach value of `lambda`

```
model_df %>%
    group_by(lambda) %>%
    summarise(mse = mean(mse)) %>%
    ggplot(aes(x = lambda, y = mse)) +
    geom_point() +
    geom_line()
```



The effect of shrinkage of *lambda* on the coefficients

```
model_df %>%
    unnest(model_fit) %>%
    group_by(lambda, variable) %>%
    summarise(mean_estimate = mean(estimate)) %>%
    ggplot(aes(x = lambda, y = mean_estimate, color = variable)) +
    geom_line() +
    geom_point()
```



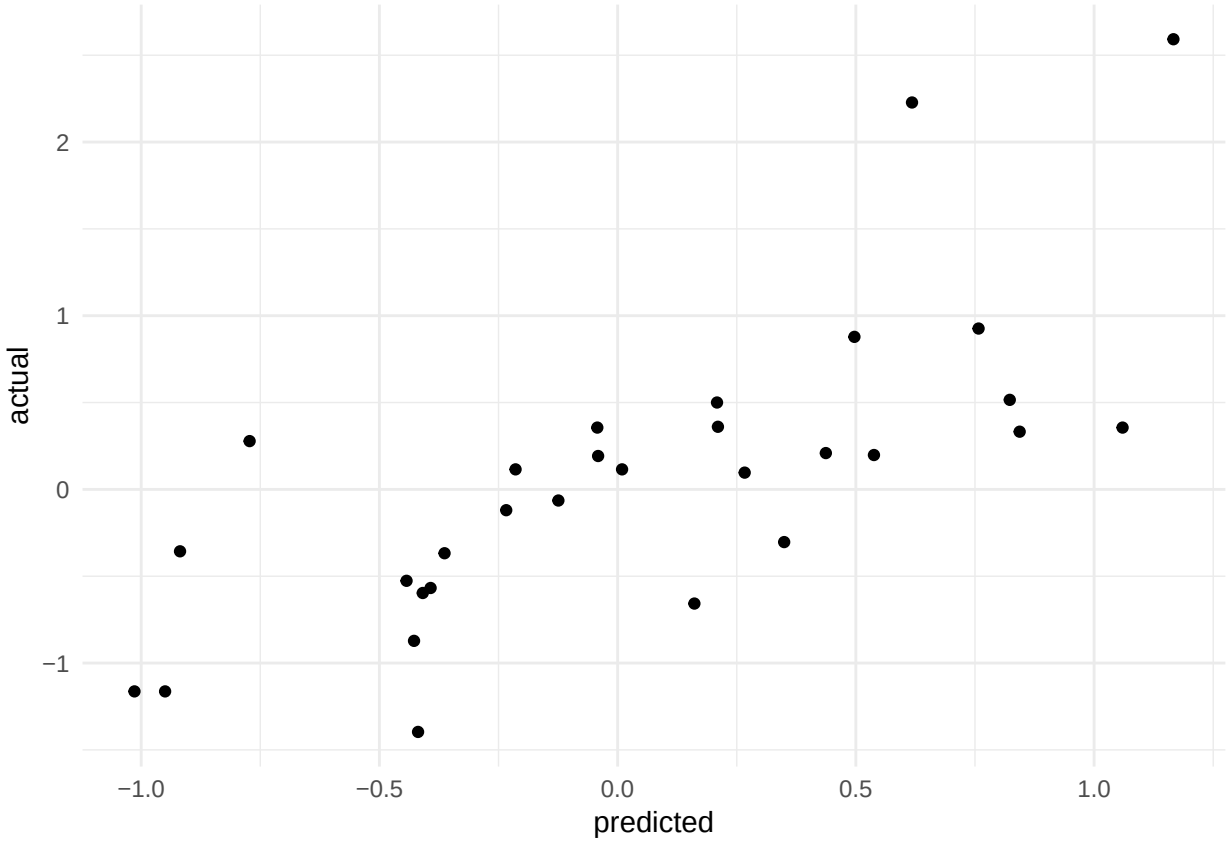
Or simply find the optimal *lambda*

```
best_lambda <- model_df %>%
    group_by(lambda) %>%
    summarise(mse = mean(mse)) %>%
    top_n(1, -mse) %>%
    pull(lambda)
best_lambda
```

```
## [1] 5.994843
```

and plot predicted against observed for the test data and compute test mse fo this choice of `lambda`

```
best_model <- lm.ridge(prostate_prep_train, formula, best_lambda)
prostate_prep_test %>% mutate(predicted = predict.ridge(., best_model, formula),
                              actual = model.frame(formula, .) %>%
                                  model.extract("response")) %>%
    ggplot(aes(x = predicted, y = actual)) +
    geom_point()
```




```
prostate_prep_test %>% mutate(predicted = predict.ridge(., best_model, formula),
                              actual = model.frame(formula, .) %>%
                                  model.extract("response")) %>%
    summarise(mse = mean((predicted - actual)^2))
```

```
## # A tibble: 1 x 1
##     mse
##   <dbl>
## 1 0.337
```

Task 2

Change the random seed to your date of birth (yymmdd). Rerun the analysis and compare the figure of mse versus lambda with the one in this sheet, you may need to adapt `lambda_seq` *in order to fit the minimum within the range of lambdas. Does the optimal lambda seem to be sensitive to the random splitting in folds? Plot the mse for each fold (rather than the average) in the same figure and report any conclusions.*

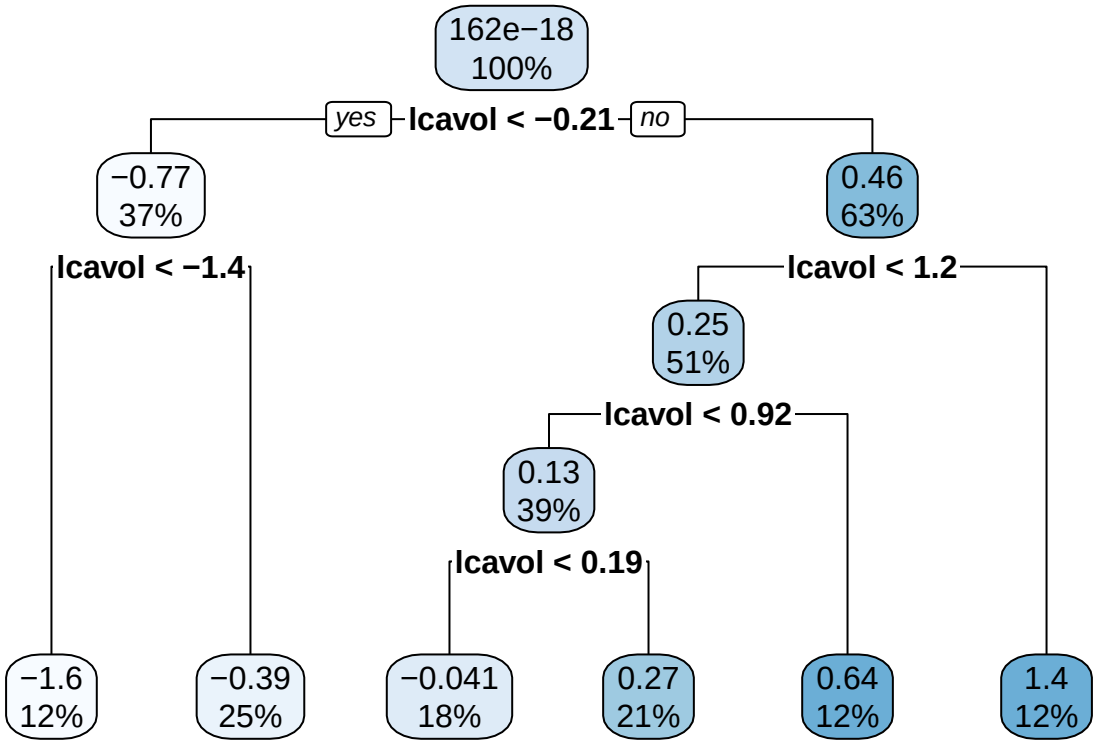
Trees

With the function `rpart` in the `rpart` library you can grow a regression tree. For a simple example we just use `lcavol` as explanatory variable

```
fit1 <- rpart(lpsa ~ lcavol, data = prostate_prep_train, control = rpart.control(cp = 0))
fit1
```

```
## n= 67
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 67 66.000000  1.623908e-16
##    2) lcavol< -0.2114333 25 17.010450 -7.702779e-01
##      4) lcavol< -1.442187 8  3.683884 -1.578363e+00 *
##      5) lcavol>=-1.442187 17  5.644197 -3.900026e-01 *
##    3) lcavol>=-0.2114333 42 25.327060  4.584987e-01
##      6) lcavol< 1.189579 34 14.593180  2.452781e-01
##       12) lcavol< 0.9240041 26 11.141090  1.253384e-01
##         24) lcavol< 0.1917455 12  3.522162 -4.092445e-02 *
##         25) lcavol>=0.1917455 14  7.002877  2.678495e-01 *
##       13) lcavol>=0.9240041 8  1.862493  6.350820e-01 *
##      7) lcavol>=1.189579 8  2.618727  1.364686e+00 *
```

```
rpart.plot(fit1)
```

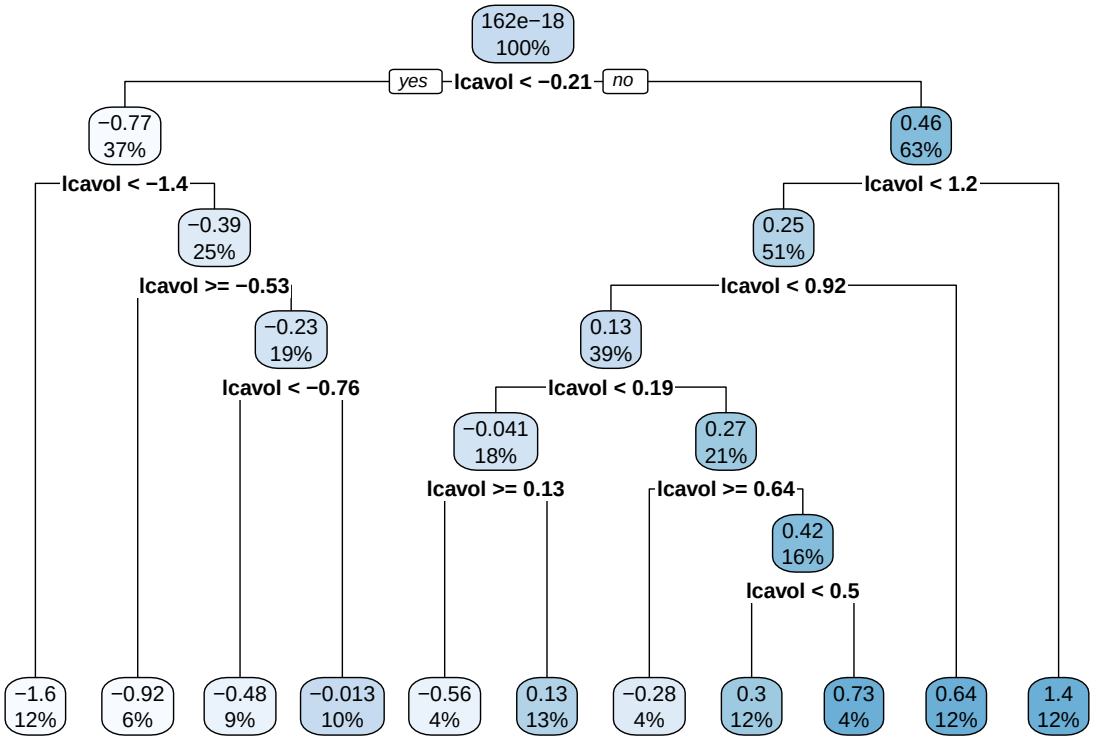


This gives six terminal nodes, since by default growth stops when there are less than 20 observations in a node (setting `cp = 0` avoids premature pruning). Using `rpart.control` we can set our own stopping criteria

```r
fit2 <- rpart(lpsa ~ lcavol, data = prostate_prep_train, control = rpart.control(minsplit = 10, cp = 0))
fit2
```

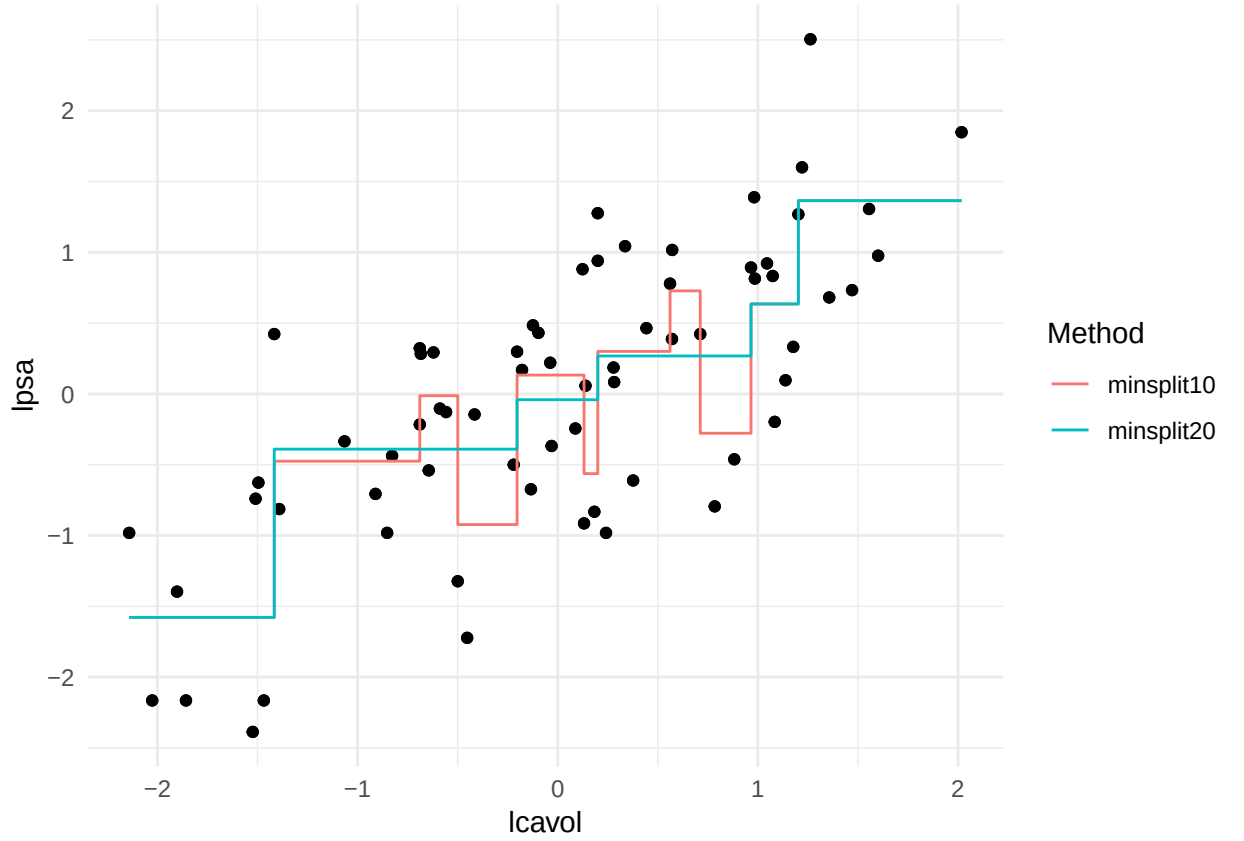
```
## n= 67
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##   1) root 67 66.0000000  1.623908e-16
##     2) lcavol< -0.2114333 25 17.0104500 -7.702779e-01
##       4) lcavol< -1.442187 8  3.6838840 -1.578363e+00 *
##       5) lcavol>=-1.442187 17  5.6441970 -3.900026e-01
##        10) lcavol>=-0.5284362 4  1.5839220 -9.226766e-01 *
##        11) lcavol< -0.5284362 13  2.5760880 -2.261029e-01
##          22) lcavol< -0.7582464 6  1.2508970 -4.750913e-01 *
##          23) lcavol>=-0.7582464 7  0.6343870 -1.268425e-02 *
##     3) lcavol>=-0.2114333 42 25.3270600  4.584987e-01
##       6) lcavol< 1.189579 34 14.5931800  2.452781e-01
##        12) lcavol< 0.9240041 26 11.1410900  1.253384e-01
##          24) lcavol< 0.1917455 12  3.5221620 -4.092445e-02
##            48) lcavol>=0.1279343 3  0.5809342 -5.630906e-01 *
##            49) lcavol< 0.1279343 9  1.8505980  1.331309e-01 *
##          25) lcavol>=0.1917455 14  7.0028770  2.678495e-01
##            50) lcavol>=0.6420237 3  0.7911555 -2.778638e-01 *
##            51) lcavol< 0.6420237 11  5.0746560  4.166804e-01
##              102) lcavol< 0.5021735 8  4.4742420  3.000914e-01 *
##              103) lcavol>=0.5021735 3  0.2016865  7.275841e-01 *
##        13) lcavol>=0.9240041 8  1.8624930  6.350820e-01 *
##       7) lcavol>=1.189579 8  2.6187270  1.364686e+00 *
```

```r
rpart.plot(fit2)
```



which grows a larger tree. We may use `predict` to visualise the step functions fitted

```
prostate_prep_train %>% mutate(minsplit20 = predict(fit1),
                    minsplit10 = predict(fit2)) %>%
    gather(key = Method, value = Predicted, minsplit20:minsplit10) %>%
    ggplot() + geom_point(aes(x = lcavol, y = lpsa)) +
    geom_step(aes(x = lcavol, y = Predicted, color = Method))
```



Evidently, the large tree (`minsplit = 10`) overfits data. Pruning of trees is done with `prune`, which minimises the cost complexity criterion (eq. (9.16) in ESL) for a given complexity parameter α =`cp`. Pruning with `cp = 0.1` gives the smaller

```
prune(fit2, cp = 0.1)
```

```
## n= 67
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
## 1) root 67 66.000000  1.623908e-16
##   2) lcavol< -0.2114333 25 17.010450 -7.702779e-01
##     4) lcavol< -1.442187 8  3.683884 -1.578363e+00 *
##     5) lcavol>=-1.442187 17  5.644197 -3.900026e-01 *
##   3) lcavol>=-0.2114333 42 25.327060  4.584987e-01
##     6) lcavol< 1.189579 34 14.593180  2.452781e-01 *
##     7) lcavol>=1.189579 8  2.618727  1.364686e+00 *
```

Task 3

Find a (near) optimal value for `cp` *using cross-validation by applying techniques as in Taks 2 to the* `cv_prostate` *data.frame. Use all variables (*`formula = lpsa ~ .`*) rather than just* `lpsa` *above and compare test mean squared error with that of the ridge regression.*

Tree bias and variance

Simple decision trees are known for their high variance and small bias. In this part we will look further into this issue by a Monte-Carlo study. In particular, we will look at variance and bias of an estimator $\hat{f}(x)$ of $f(x)$ for various values of fixed x. We will do so by

- Choosing a (non-constant) function $f(x)$, $0 \leq x \leq 1$, distributions for the input variable X and the observation error ϵ, and a sample size n.
- Simulate N samples of size n from the distributions of X and $Y = f(X) + \epsilon$.
- For each of the N samples, fit $\hat{f}$, and compute its values on a grid.
- Approximate bias/variance for each value on the grid by averaging over the N samples.

Here is a simple example, approximating bias, variance and mse of a polynomial regression applied to $f(x) = \sin(5x)$. Note that we are examining performance as a function of x for fixed hyperparameters, rather than mean (over x) performance as a function of hyperparameter as before.

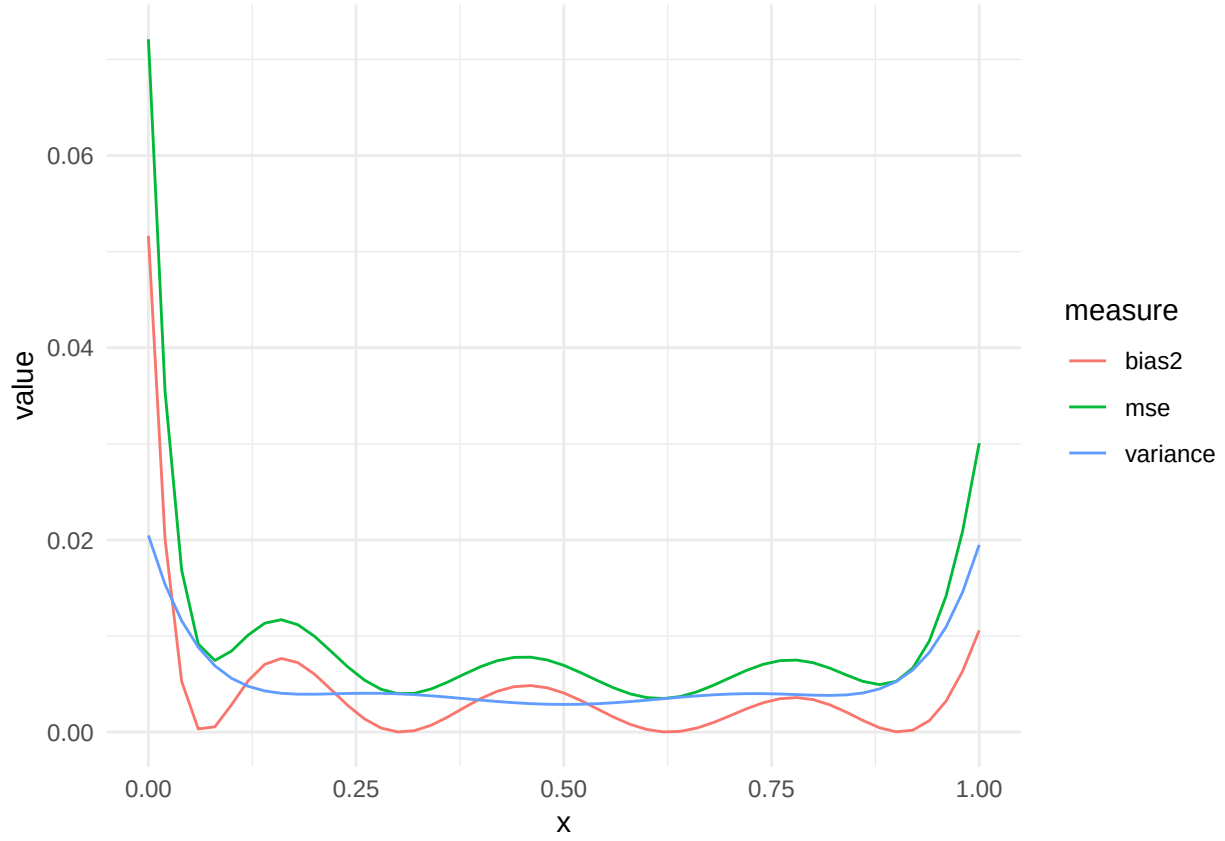
```
# The function to estimate
f <- function(x){
    sin(x * 5)
}

# A function to simulate a sample of size n (uniform X)
sim_data <- function(n = 100, f. = f, sd = 1/3){
    data.frame(x = runif(n)) %>% mutate(y = f(x) + rnorm(n, sd = sd))
}

# Define a grid of points for which
# performance shoudl be evaluated
newdata <- data.frame(x = 0:50/50)

# Number of Monte-Carlo samples
N <- 100
```

```
tibble(data = rerun(N, sim_data())) %>% # Draw N samples
    # Fit a cubic polynomial
    mutate(fit = map(data, ~lm(y ~ poly(x, 3), data = .x)),
           predicted = map(fit, ~mutate(newdata, predicted = predict(.x, newdata = newdata)))) %>%
    unnest(predicted, .id = "name") %>%
    group_by(x) %>%
    summarise(bias2 = mean(f(x) - predicted)^2,
              variance = var(predicted),
              mse = bias2 + variance) %>%
    gather(key = "measure", value = "value", -x) %>%
    ggplot(aes(x = x, y = value, color = measure)) +
    geom_line()
```



For this example (fitting a cubic polynomial to a sine), squared bias and variance seems fairly well balanced.

Task 4

Pick your own function and distribution as above and repeat the analysis, fitting an unpruned decision tree using ***rpart*** *instead of a cubic (try a few values of* ***minsplit****). Do the trees balance variance and squared bias well?*

Task 5

Bagging can be used to reduce variance of trees. It works by averaging trees applied to boostrap samples of data. Write a function

```
bag_tree <- function(data, newdata = data.frame(x = 0:50/50), B = 10){
    ...
}
```

that takes ***data*** *as input, draws* ***B*** *Bootstrap resamples (draw with replacement) from data, fits a decision tree (using* ***rpart****) to each resample and predicts values at* ***newdata****. The function should then return a* ***data.frame*** *with columns* ***x*** *and* ***predicted****, where* ***predicted*** *is the average of the resampled predictions. Finally, repeat task 4 with this new function.*